Hierarchical Software-Hardware Prefetching

CS683 Project Checkpoint 2

Team Members
Bala Murugan S(24M1173), Gautam Govindaraju(24M1166), Debaditya Mohanty(24M1147)
24m1173@iitb.ac.in, 24m1166@iitb.ac.in, 24m1147@iitb.ac.in
TEAM NAME: Cache Catalysts

November 23, 2025

Problem Statement

- Instruction Cache Misses are very frequent in server workloads due to large number of instructions footprints and irregular control flows.
- Predicting Access Patterns in these workloads is quite difficult because of less repetition frequency.
- Traditional Hardware Prefetchers like Stride and Next N line Prefetchers are often insufficient in server applications.
- Software-only Prefetching has high overhead and often lacks adaptability.
- A combined Hardware-Software Approach promises better timeliness, accuracy, and coverage

Goal of the Project

- Hierarchichal Software-Hardware Instruction Prefetcher proposes a cooperative software-hardware approach that partitions applications into coarse-grained ”Bundles” records instruction footprints at runtime, and replays them on future executions to improve prefetch accuracy, coverage, timeliness, etc.
- The Goal of this project is to implement this prefetcher and verify the performance of coarse-grained bundle based instruction prefetching compared with traditional prefetchers on server workloads.(request parsers, database simuulators, etc)

Plan for Checkpoint 1 and Division of Labour

- In Checkpoint 1, we will do **Static Analysis and Bundle Identification** which includes the following tasks.
- Use LLVM/Clang to compile program into LLVM or IR s(Intermediate Representation of LLVM) (Gautam Govindaraju)
- Construction of function call graph from compiled LLVM file. (Bala Murugan S)
- Estimation of function + reachable code size for each node.
- Select bundle entry points and creation and filling of bundles.json for future use. (Debaditya Mohanty)
- So, in checkpoint 1, we are planning to show how we constructed the call graph and creation of bundles.json.

Server Workload Simulator - C to LLVM

- LLVM is a compiler framework that translates high-level code into a platform-independent **Intermediate Representation (IR)** for analysis and optimization.
- The IR exposes detailed instruction-level structure, enabling static analysis such as call-graph construction and Bundle identification for prefetching.
- We have wriiten a C file which is named as **server_sim.c** which is a Multi-threaded synthetic HTTP-like server, It Simulates request parsing, authentication, business logic, in-memory DB access, and logging.
- This file is compiled to LLVM IR using Clang for static analysis.
- Each functional unit of this code represents a logical code region for Bundle formation.

Construction of Call Graph from LLVM

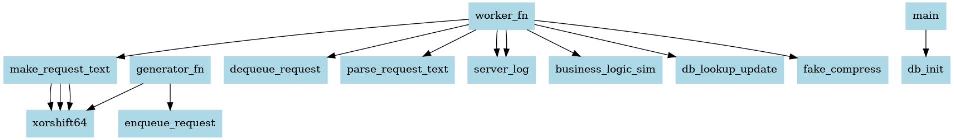
- A **Call Graph** represents function call relationships ($\textit{Caller} \rightarrow \textit{Callee}$) within a program.
- The above described call graph generation and Bundle Formation is implemented in the `callgraph_bundle.cpp`.
- LLVM IR is parsed using `parseIRFile()` to obtain all functions and instructions.
- The `CallGraph` class builds caller–callee mappings and instruction counts for each function.
- Results are exported as `callgraph.txt` and `callgraph.dot` for visualization.
- The DOT graph is converted to an image using Graphviz command: `dot -Tpng callgraph.dot -o callgraph.png`.

Creation of `bundles.json` File

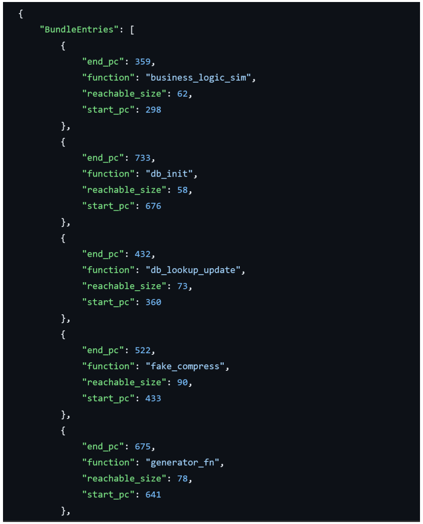
- Bundles represent coarse functional regions derived from the LLVM Call Graph.
- Each instruction is assigned a unique ID to define `start_pc` and `end_pc`.
- Reachable code size = no.of.instructions in that function + no.of.instructions in the functions it calls.
- Functions exceeding a threshold are selected as **Bundle Entry Points**.
- We can change this threshold, so that even smaller functions could be recorded during this phase and prefetched later during the replay phase.
- Bundle metadata (name, reachable size, start/end PCs) is stored in `bundles.json`.
- Thus the Execution of **callgraph_bundle.cpp** over the **server_sim.ll** file gives out the `callgraph.txt`, `callgraph.dot`, and `bundles.json` files as output.

Generated Call Graph Image

- The Call Graph shows direct caller-callee relationships extracted from the LLVM IR, where each node is a function and each arrow denotes an explicit function call.
- The **main()** function directly calls only `db_init()`, while thread entry functions such as `worker_fn` and `generator_fn` appear as separate nodes since they are launched indirectly via `pthread_create()`. Then these functions call other functions directly which are visible in the below generated image.



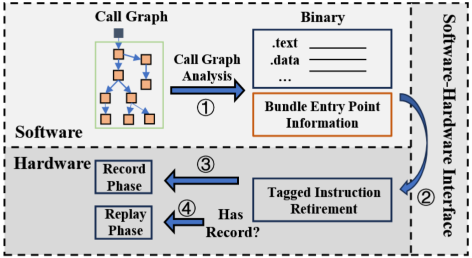
Generated `bundle.json` File



```
{
    "BundleEntries": [
        {
            "end_pc": 359,
            "function": "business_logic_sim",
            "reachable_size": 62,
            "start_pc": 298
        },
        {
            "end_pc": 733,
            "function": "db_init",
            "reachable_size": 58,
            "start_pc": 676
        },
        {
            "end_pc": 432,
            "function": "db_lookup_update",
            "reachable_size": 73,
            "start_pc": 360
        },
        {
            "end_pc": 522,
            "function": "fake_compress",
            "reachable_size": 90,
            "start_pc": 433
        },
        {
            "end_pc": 675,
            "function": "generator_fn",
            "reachable_size": 78,
            "start_pc": 641
        },
```

Block Diagram

- The below block diagram from the ASPLOS research paper represents the flow of this project clearly.
- In Checkpoint 1, we have implemented the **Call Graph Analysis** and **Bundle Formation**.
- In Checkpoint 2, we will implement the **Bundle based Prefetcher** and compare its performance with the traditional Prefetchers.



Plan for Checkpoint 2 and Division of Labour

- We will develop a **Bundle-Based Prefetcher** within the **Gem5 simulator** that uses the `bundles.json` metadata generated from Checkpoint-1. (Gautam Govindaraju)
- Run the `server_sim.c` workload on Gem5 and compare its performance using a **traditional hardware prefetcher** versus the proposed **Bundle-based prefetcher**.
- Record metrics such as IPC, instruction-cache MPKI, and speedup to quantify improvements. (Bala Murugan S)
- Extract Bundles dynamically from the **Gem5 workload trace files**, update them into a new `bundles.json`, and re-run the prefetcher using this trace-driven metadata. (Debaditya Mohanty)
- Perform a comparative analysis between the traditional prefetchers and the proposed prefetcher to evaluate prefetcher accuracy and efficiency.

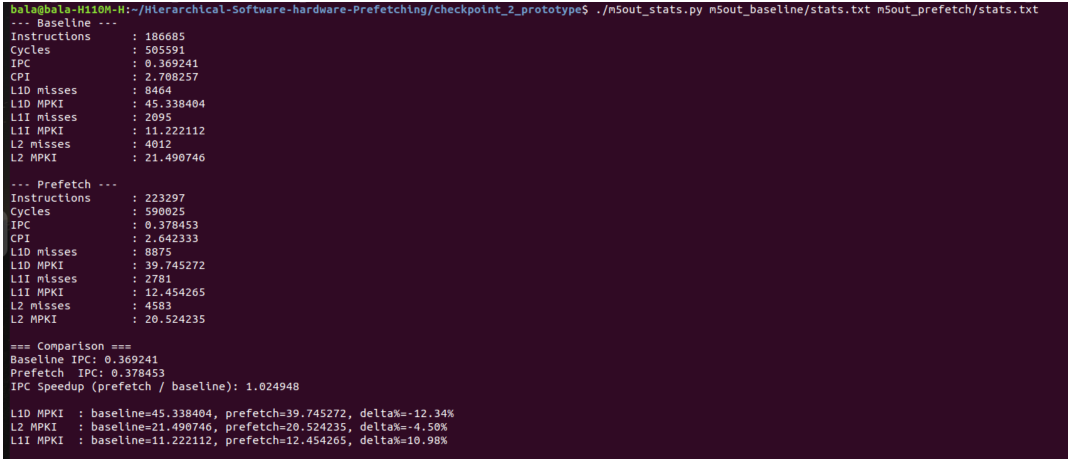
Bundle Mapping and Replay Setup

- We used a Python script `map_bundles_to_pcs.py` to generate PC in form of hexadecimal addresses from the `bundles.json` file by comparing it with `server_sim.o` file.
- Each bundle stored a function name with associated `start_pc` and `end_pc`, these PCs are in hexadecimal format.
- These Bundles are saved as `bundles_pc.json` and loaded at runtime.
- Then we implemented `bundle_replay.c` to parse JSON into a global bundle table.
- We also modified the file as `server_sim_prefetch.c`, and the modifications triggered prefetches by calling `bundle_prefetch_by_name(...)` inside the functions of `server_sim.c`.

Faced Limitations

- The Prefetching used in the `bundle_replay.c` is page-level `madvise()` and `__builtin_prefetch()` per page.
- The Bundle lookup strategy we used was linear string-match, it was adding overhead at each call.
- The prefetch region spanned the full PC range, so it was often covering unused pages, which will not be used in the near future.
- Several cold regions were unnecessarily prefetched, causing cache pollution.
- Improvement was small, around 2% IPC gain, limited by over-prefetching.
- Then we had a plan of converting the `server_sim.c` to trace files using gem5. Then we thought of rebuilding gem5 by adding the prefetcher files `bundle_prefetcher.cc` and `bundle_prefetcher.h`.

Command-Line Output (Baseline vs Prefetch)



```
bala@bala-H110M-H:~/Hierarchical-Software-hardware-Prefetching/checkpoint_2_prototype$ ./m5out_stats.py m5out_baseline/stats.txt m5out_prefetch/stats.txt
--- Baseline ---
Instructions      : 186685
Cycles            : 505591
IPC               : 0.369241
CPI               : 2.708257
L1D misses        : 8464
L1D MPKI          : 45.338404
L1I misses        : 2095
L1I MPKI          : 11.222112
L2 misses         : 4012
L2 MPKI           : 21.490746

--- Prefetch ---
Instructions      : 223297
Cycles            : 590025
IPC               : 0.378453
CPI               : 2.642333
L1D misses        : 8875
L1D MPKI          : 39.745272
L1I misses        : 2781
L1I MPKI          : 12.454265
L2 misses         : 4583
L2 MPKI           : 20.524235

=== Comparison ===
Baseline IPC: 0.369241
Prefetch  IPC: 0.378453
IPC Speedup (prefetch / baseline): 1.024948

L1D MPKI  : baseline=45.338404, prefetch=39.745272, delta%=-12.34%
L2 MPKI   : baseline=21.490746, prefetch=20.524235, delta%=-4.50%
L1I MPKI  : baseline=11.222112, prefetch=12.454265, delta%=10.98%
```

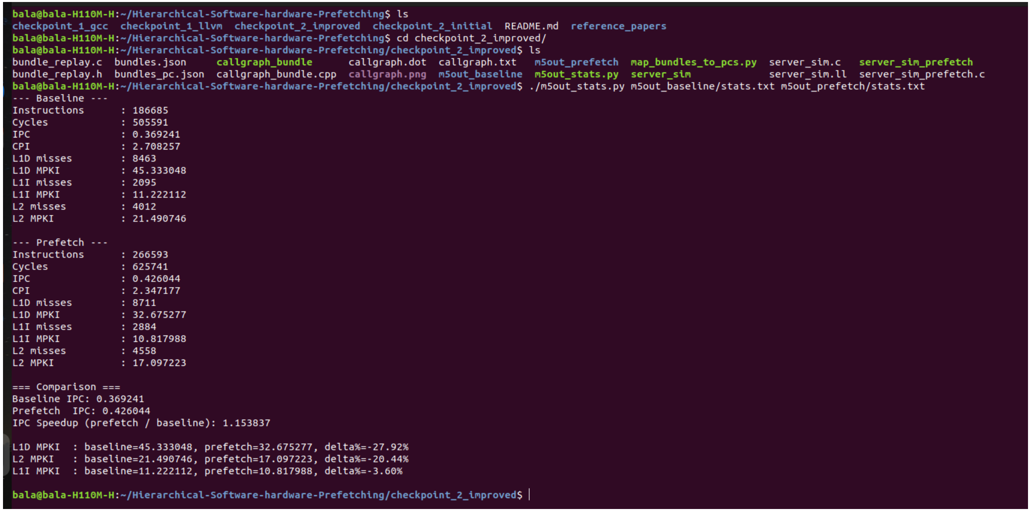
Attempted gem5 Trace-Based Bundle Prefetcher

- We attempted a hardware-style `bundle_prefetcher.cc` inside gem5, requiring exact instruction PCs from elastic trace files.
- The plan was to run `server_sim.c` once, extract .tr traces, and map committed PCs to function bundles for precise prefetch triggers.
- gem5.fast rejected key trace options, causing probe-listener type errors, because O3CPU trace probes require strict DerivO3 trace support.
- Elastic-trace decoding failed due to missing proto definitions, and gem5 SE mode produced empty or incompatible trace files.
- Due to repeated model incompatibilities and unstable tracing, we reverted to user-space prefetching with an improved prefetching strategy.

Improved Prefetching Strategy

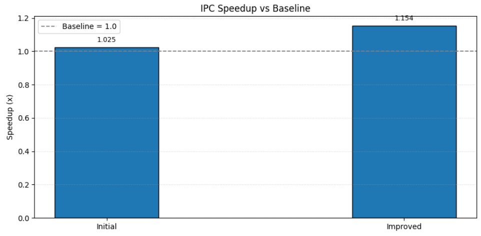
- We replaced broad PC-range prefetching with page-aligned data prefetching, targeting only memory regions actually used by each bundle.
- Then introduced selective bundle triggers at function boundaries, mapping names directly to hot bundles from `bundles_pc.json`.
- This limited prefetch size by capping total pages avoided cold bundles, reducing cache pollution and unnecessary memory traffic.
- Then we combined `madvise()` with `__builtin_prefetch`, providing both OS-level and CPU-level prefetch hints efficiently.
- The above said modifications to `bundle_replay.c` and `server_sim.c` gave a final IPC improvement of 15% compared to the baseline.

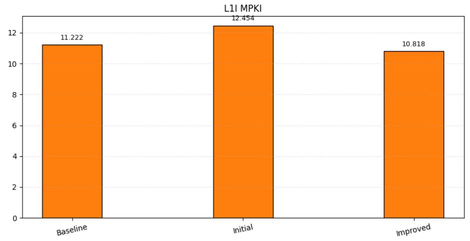
Command Line Output(Baseline vs Improved Prefetch)

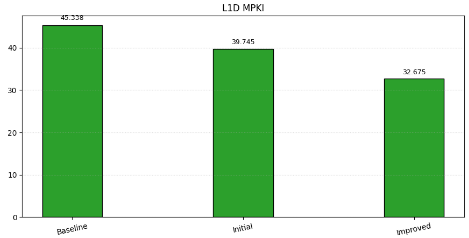


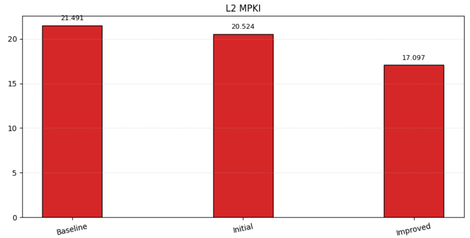
```
bala@bala-H110M-H:~/Hierarchical-Software-hardware-Prefetching$ ls
checkpoint_1_gcc  checkpoint_1_llvm  checkpoint_2_improved  checkpoint_2_initial  README.md  reference_papers
bala@bala-H110M-H:~/Hierarchical-Software-hardware-Prefetching$ cd checkpoint_2_improved/
bala@bala-H110M-H:~/Hierarchical-Software-hardware-Prefetching/checkpoint_2_improved$ ls
bundle_replay.c  bundles.json     callgraph_bundle      callgraph.dot  callgraph.txt  m5out_prefetch  map_bundles_to_pcs.py  server_sim.c   server_sim_prefetch
bundle_replay.h  bundles_pc.json  callgraph_bundle.cpp  callgraph.png  m5out_baseline  m5out_stats.py  server_sim             server_sim.ll  server_sim_prefetch.c
bala@bala-H110M-H:~/Hierarchical-Software-hardware-Prefetching/checkpoint_2_improved$ ./m5out_stats.py m5out_baseline/stats.txt m5out_prefetch/stats.txt
--- Baseline ---
Instructions      : 186685
Cycles            : 505591
IPC               : 0.369241
CPI               : 2.708257
L1D misses        : 8463
L1D MPKI          : 45.333048
L1I misses        : 2095
L1I MPKI          : 11.222112
L2 misses         : 4012
L2 MPKI           : 21.490746

--- Prefetch ---
Instructions      : 266593
Cycles            : 625741
IPC               : 0.426044
CPI               : 2.347177
L1D misses        : 8711
L1D MPKI          : 32.675277
L1I misses        : 2884
L1I MPKI          : 10.817988
L2 misses         : 4558
L2 MPKI           : 17.097223

=== Comparison ===
Baseline IPC: 0.369241
Prefetch  IPC: 0.426044
IPC Speedup (prefetch / baseline): 1.153837

L1D MPKI  : baseline=45.333048, prefetch=32.675277, delta%=-27.92%
L2 MPKI   : baseline=21.490746, prefetch=17.097223, delta%=-20.44%
L1I MPKI  : baseline=11.222112, prefetch=10.817988, delta%=-3.60%

bala@bala-H110M-H:~/Hierarchical-Software-hardware-Prefetching/checkpoint_2_improved$
```

Baseline Vs Initial Prefetching Vs Improved Prefetching









Conclusion

- We began by compiling the program to LLVM IR and generating the complete call graph to identify bundle entry functions.
- Using this graph, we constructed `bundles.json` and built an initial replay-based prefetch system using function-level triggers.
- Our first gem5-based bundle prefetcher attempt faced tracing issues, including incompatible probes and unstable elastic-trace decoding.
- We reverted to a robust user-space prefetch strategy using page-level `madvise` and a targeted `__builtin_prefetch`.
- In future, we plan to revisit gem5 with a correct trace pipeline and fully integrate a hardware-like bundle prefetcher for evaluation.

Reference Papers

- T. Zhang et al., ”Hierarchical Prefetching: A software-hardware instruction prefetcher for server applications,” in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25)*, Vol. 2, Rotterdam, Netherlands, Mar. 30–Apr. 3, 2025, pp. 529–544. doi: 10.1145/3676641.3716260. (Main Research Paper we are implementing this)
- M. Annavaram, J. M. Patel, and E. S. Davidson, ”Call graph prefetching for database applications,” *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, Monterrey, Mexico, 2001, pp. 281–290. doi: 10.1109/HPCA.2001.903270.
- G. Reinman, B. Calder, and T. Austin, ”Fetch directed instruction prefetching,” *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-32)*, Haifa, Israel, 1999, pp. 16–27. doi: 10.1109/MICRO.1999.809439.

Github Link

- https://github.com/bala-murugan-03/Hierarchical-Software-hardware-Prefetching.git

Thank You!